PLOR: General Transactions with Predictable, Low Tail Latency

Youmin Chen, Xiangyao Yu[†], Paraschos Koutris[†],
Andrea C. Arpaci-Dusseau[†], Remzi H. Arpaci-Dusseau[†], Jiwu Shu[*]
Tsinghua University, [†]University of Wisconsin - Madison

ABSTRACT

We present *pessimistic locking and optimistic reading* (PLOR), a hybrid concurrency control protocol for in-memory transaction systems that delivers high throughput and low tail latency. PLOR is especially designed for high-contention workloads: for high throughput, transactions are allowed to access records without being blocked by lock conflicts in the read phase; for low tail latency, conflict detection is delayed to the commit phase, where old transactions are always committed first using the timestamps in the lock. We demonstrate the efficacy of this approach under a variety of setups (e.g., stored-procedures, interactive mode, and persistent logging, etc.). Experiments show that PLOR delivers close or comparable throughput to that of Silo and TicToc in stored-procedures, while reducing 99.9th percentile latency by 8.8× to 14.5×. In the interactive processing mode, PLOR even achieves up to 2× higher throughput.

CCS CONCEPTS

• **Information systems → Database transaction processing**.

KEYWORDS

OLTP, concurrency control, two-phase locking, tail latency

ACM Reference Format:
Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, NY, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3517879

1 INTRODUCTION

With the increasing popularity of multi-core servers and large-capacity main memories, database research has seen a renaissance over the past decade [12, 22, 23, 44, 49, 55, 56]. By distributing the whole data in DRAM and incorporating light-weight concurrency control protocols, these main memory databases achieve impressive performance. Recently, latency-critical applications further require data services to deliver low and predictable latency [11, 30, 46]. For large-scale systems serving web search, email and many other types of interactive services, data queries are fanned out to thousands of data servers when processing a single request, where predictable and low *tail latency* is extremely important.

Many recent efforts to cut tail latencies focused on various layers in the operating system, including core scheduling [20, 29, 35, 36], queue management [13], and caching mechanisms [6]. They prevent latency spikes either by separating large requests from small ones, thus avoiding the queuing delay caused by head-of-line blocking (HLB), or by minimizing the inferences of background operations (e.g., garbage collection, data compaction, etc.).

Unfortunately, scant attention has been paid to the impact of *request conflicts* on tail latencies. Conflicts are extremely common in modern highly contentious transactional workloads: at Twitter [53], for example, 25% of the clusters have a write ratio of more than 50% and most of them follow the Zipfian distribution with skewness in the range from 1 to 1.25 (highly skewed); similar workloads also exist at Facebook [8]. While a vast amount of research has been devoted to mitigating the overhead brought by request conflicts, e.g., transaction chopping [40, 57], transaction scheduling [14, 38, 43], program analysis [27, 50, 51], etc., such techniques mainly focus on the throughput (i.e., txn/s), and do *not* consider tail latencies.

In this paper, we first study how existing concurrency control protocols, e.g., optimistic concurrency control (OCC) and pessimistic two-phase locking (2PL), function in terms of throughput and tail latency. At one extreme, OCC assumes conflicts between transactions are rare, so it detects conflicts only at the commit phase, and thus incurs less locking overhead and scales effectively on multi-core servers. Besides, in-memory transactions are typically *short-lived*, enabling OCC to be efficient even at a high abort rate since the overhead of re-running aborted transactions is not high [19]. Actually, most recent work focuses on the variants of OCC to deliver high performance (e.g., Silo [44], FOEDUS [23], MOCC [49], and T/S ordering protocols such as Cicada [25] and TicToc [55]). At the other extreme is 2PL, which requires a transaction to acquire locks before accessing records. 2PL's long locking periods and requirements for read locking make it less attractive for high throughput environments, and thus it is seldom adopted in recent in-memory databases. However, when we run the default YCSB-A workload (high contention, r:50% and w:50%), some 2PL variants (e.g., WOUND_WAIT), exhibit a 99.9th percentile latency that is 12 − 20× lower than that of Silo [44], a representative OCC-based DBMS.

We observe that the tail latency and throughput of OCC and 2PL are hurt by *independent factors*. For the tail latency, we notice that transactions with high latencies typically abort many times. WOUND_WAIT resolves conflicts by ensuring that transactions with older timestamps are always committed first. Since aborted transactions still use old timestamps, they typically have a higher priority to commit. As a result, WOUND_WAIT delivers low tail latency by preventing aborted transactions from aborting again. For the throughput, 2PL is less efficient since it incurs unnecessary blocking and locking overheads when transactions conflict.

[*]Jiwu Shu (shujw@tsinghua.edu.cn) is the corresponding author.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517879

Intuitively, with a proper combination of OCC and 2PL, we can achieve both high throughput and low tail latency. We explore this hypothesis by introducing the notion of *pessimistic locking and optimistic reading*, or PLOR. Specifically, a transaction must acquire locks for both reads and writes before accessing records from the database (i.e., *pessimistic locking*); but the transaction can ignore lock conflicts and access records directly without being blocked (i.e., *optimistic reading*).

PLOR processes a transaction with the standard read and commit phases. In the **read** phase, the transaction acquires a read or write lock by storing its state (including the timestamp) in the lock manager, which helps enforce the commit priority between conflicting transactions when they commit. PLOR does not check lock conflicts in the read phase, so the locking process finishes immediately. PLOR delays the detection of conflicts to the commit phase, where we need to ensure that conflicts can be *safely* delayed without causing inconsistency, and *correctly* resolved without violating the isolation guarantee. To achieve this, PLOR requires writing transactions to buffer updates locally in the read phase, during which a reader can read this record directly without being blocked; similarly, writes can also bypass reads since read operations do not modify the database anyway. In the **commit** phase, PLOR resolves potential conflicts by scanning the transaction states that reside in all acquired locks – PLOR kills the corresponding transaction if its timestamps are higher; otherwise, PLOR lets the committing transaction wait until the conflicting one has committed. Overall, PLOR reads records optimistically in the read phase to remove the unnecessary blocking overhead (for high throughput); in the commit phase, PLOR ensures that transactions are always committed with the timestamp order (for low tail latency).

Unlike OCC that only requires write locks, transactions in PLOR require locks for both read and write operations. We incorporate a *latch-free locker* to minimize the locking overhead. We find, interestingly, decomposing the lock procedure into separate steps (i.e., lock acquisition and conflict detection) facilitates the design of a light-weight lock primitive using lock-free data structures (§4.2).

PLOR provides the *serializability* isolation by default, despite it can be easily adapted to provide weaker isolation levels. For example, when running `Stock-Level` transactions of TPC-C, PLOR only ensures the *read-committed* isolation. We evaluate PLOR under a variety of setups to explore the design space and trade-off of PLOR, including the transaction processing model (e.g., *stored-procedures* vs. *interactive transactions*), data persistence (by logging data to Optane DC Persistent Memories [28]), etc. In stored-procedure mode, PLOR only underperforms Silo and TicToc by 9% to 19% with the YCSB-A workload, and achieves comparable throughput with the TPC-C workload. Importantly, PLOR reduces 99.9th percentile latency by an order of magnitude. With the interactive mode, PLOR even delivers up to 2× higher throughput.

2 BACKGROUND AND MOTIVATION

2.1 Two-Phase Locking

Two-phase locking (2PL), as the first method proved to ensure the correct execution of concurrent transactions, has been widely adopted in traditional DBMSs. 2PL requires transactions to acquire the lock in either exclusive or shared mode for a record before writing or reading that record. To ensure correctness, 2PL enforces two rules: first, different transactions cannot own *conflicting locks* simultaneously; second, once a transaction surrenders ownership of a lock, it cannot obtain additional locks. The second rule decouples 2PL into two phases (i.e., *growing phase* and *shrinking phase*). When a transaction fails to acquire the lock due to the violation of the first rule, 2PL puts the requesting transaction in the waiting queue until the lock is available. A number of 2PL variants exist to avoid deadlocks in case of cycles of waiting:

NO_WAIT never waits when acquiring a lock – whenever a lock request is denied, the transaction is aborted immediately.

WAIT_DIE is a non-preemptive technique that assigns a timestamp to each transaction and avoids deadlocks by comparing the timestamp with the lock owners. Specifically, when one transaction (e.g., T_i) requests a lock currently held by some owners, T_i is allowed to be in the waiting queue only if it has a timestamp smaller than that of all owners, namely WAIT; otherwise, T_i is aborted, namely DIE.

WOUND_WAIT is a counterpart to WAIT_DIE. When a transaction (e.g., T_i) requests a lock currently held by some owners, the owners whose timestamps are bigger than T_i are aborted, namely WOUND; then, T_i either becomes the new owner or waits for the lock, depending on whether all owners are aborted, namely WAIT.

2.2 Optimistic Concurrency Control

OCC assumes that conflicts between transactions are rare, so it processes transactions without locking records before committing them. OCC consists of three phases, including (1) *read phase:* a transaction reads records from the database and performs all updates to a local private buffer; (2) *validation phase:* the transaction checks whether the records it has read or written are modified by other concurrent transactions; and (3) *write phase:* after a successful validation, OCC commits the modified records to the database.

Silo [44] is an in-memory DBMS that follows the standard phases in OCC to process transactions. In the validation phase, Silo first locks all records in the write-set, and aborts the transaction if a deadlock is suspected. Lock acquisition is done by toggling a 64-bit word using atomic operations. Then, the transaction's timestamp is generated to mark the serialization point. It next checks that records in the read-set have not changed and are not locked by other transactions, which is also finished in a lock-free manner. In this paper, we choose Silo to represent the OCC-based systems.

2.3 Motivation

In this section, we discuss how 2PL and OCC perform in terms of throughput and tail latency. The experiment uses the YCSB benchmark and we choose the default YCSB-A workload with varying skewness, which determines the contention level. The results are shown in Figure 1, where the 99.9th percentile latencies and throughput are collected as we increase the number of threads from 1 to 36. We present the details of this experiment in Section 6 and summarize the main observations here. When conflicts are rare (in Figure 1a, $\theta = 0.5$), we find that the tail latencies of all systems are extremely low (in the range of tens of μs), which are only slightly higher than their median latencies (not shown). However, as we run the high-contention workload (in Figure 1b), their performance varies greatly in terms of both throughput and tail latency.

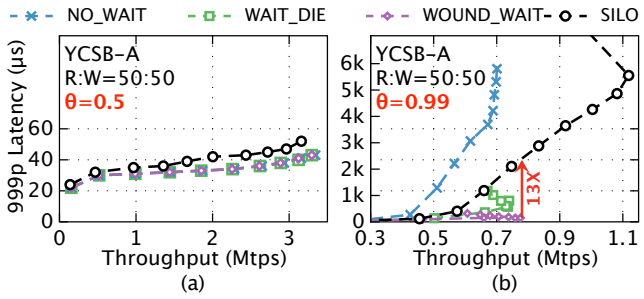

Figure 1: 2PL vs. OCC in Tail Latency and Throughput.

2.3.1 ***Throughput.*** As expected, Silo achieves the highest peak throughput among the compared schemes, which outperforms other 2PL schemes by 40% - 57%. By analyzing the protocol-level timeline of OCC and 2PL, we specify a number of factors that hurt the throughput of 2PL schemes. Figure 2 shows how 2PL and OCC handle different types of conflicts.

• *Read-write conflicts.* Under 2PL, when a transaction (i.e., $T1$ in Figure 2a) is holding an exclusive write lock, another reader (i.e., $T2$) has to block until $T1$ releases the lock. However, this is not the case in OCC since readers do not need to acquire read locks. OCC achieves this via a private buffer, where write operations never modify the database before a transaction is committed. If $T2$ finishes the validation phase before $T1$ commits, then both of them can commit (i.e., $T1$ and $T2$ in Figure 2b). Otherwise, the reader (i.e., $T2'$ in Figure 2b) aborts and reruns.

• *Write-write conflicts.* 2PL acquires write locks before accessing records, and releases them after the records have been committed. As shown in Figure 2a, $T3$ blocks $T4$ all the way due to the write conflicts, and $T4$ may further block other transactions and cause *cascading blocking*. Under OCC, write locks are acquired at the commit phase, which incur less blocking time. For read-modify-write (RMW) operations, OCC needs to abort either $T3$ or $T4$ to ensure serializability, which, instead, alleviates the cascading blocking overhead.

• *Locking overhead.* A lock primitive under 2PL contains multiple lock states (e.g., lock owners and waiters) and needs to support both shared-read and exclusive-write semantics. As such, 2PL typically suffers higher locking overhead. Silo, instead, does not need shared locks and it avoids deadlocks by acquiring write locks in a deterministic global order (e.g., pointer addresses of records). Hence, it does not need waiting queues in the lock, and can acquire a lock by simply toggling an atomic word via atomic CAS operations.

• *Read-read contention.* 2PL ensures serializability of read-only transactions by acquiring read locks, which still incurs cache coherency traffic when updating the same lock states. Silo only needs to check versions for the records in the read-set, so it never blocks other transactions (such reads are called *invisible reads* since they do not write any control information to inform other writers [33]).

2.3.2 ***Tail Latency.*** WAIT_DIE and WOUND_WAIT exhibit lower tail latency than Silo. When running at a target load of 0.7 Mtps, they can restrict their 99.9th percentile (999p) latencies within 700 μs and 200 μs, respectively, 2.6× and 13× lower than that of Silo.

We observe that the latency of a committed transaction is highly related to the number of aborts, i.e., more abort times lead to higher latency. Silo executes transactions without acquiring locks during the *read phase*, so the already aborted transactions still share the same chance as other newly invoked ones to commit during the next retry, and this causes its high tail latency. In Figure 2b, $T2'$ owns the oldest timestamp but still aborts. In some extreme cases where the writers are modifying a record repeatedly, a read in Silo may even suffer from starvation and cause long tail latency. As described in §2.1, both WAIT_DIE and WOUND_WAIT are starvation-free protocols. They ensure that transactions with smaller timestamps have a higher priority to commit, and aborted transactions typically own smaller timestamps since they start earlier. Thus, they grant aborted transactions a higher priority to commit.

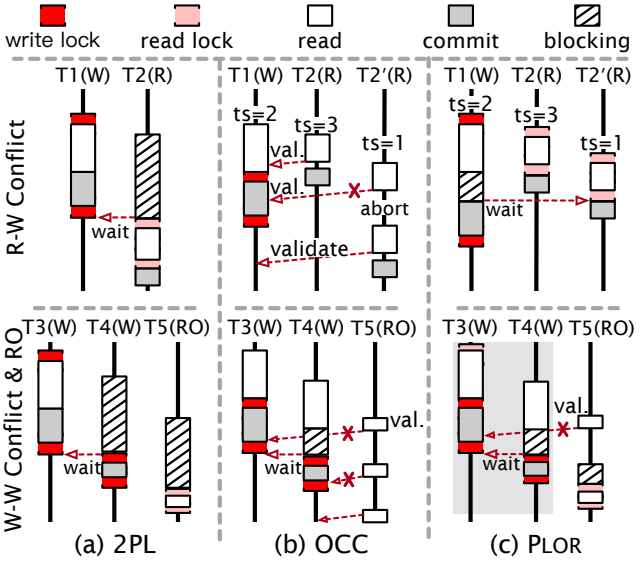

Figure 2: Analysis of 2PL and OCC. ***In (a), 2PL always causes blocking when transactions conflict; In (b), OCC has the chance to commit $T2$ without being blocked; In (c), PLOR delays conflict detection to the commit phase. Alternatively, PLOR can delay acquiring write locks to reduce w-w conflicts (gray area). PLOR uses a dynamic way to avoid read-only transactions aborting repeatedly.***

We also find that WOUND_WAIT's tail latency is much lower than that of WAIT_DIE. To understand this, we further analyze how they manage the waiting queue in a lock. In WOUND_WAIT, when the lock owner releases the lock, it grants the lock to the transaction that owns the *oldest* timestamp in the waiting queue. Hence, *WOUND_WAIT always commits transactions with the timestamp order, and thus delivers the lowest tail latency.* In WAIT_DIE, however, the lock owner grants the lock to the transaction with the largest timestamp in the waiting queue. This is because WAIT_DIE keeps an invariant that all the waiters in the queue should have smaller timestamps than the lock owner. NO_WAIT has the highest tail latency, since it aborts a transaction whenever a lock conflict occurs, regardless of how many times the transaction has been aborted before. To summarize our discussion so far, neither 2PL nor OCC achieves high throughput and low tail latency simultaneously.

3 DESIGN PRINCIPLES AND OVERVIEW

An important takeaway from §2.3 is that, a proper combination of OCC (with high throughput) and WOUND_WAIT (with low tail

latency) in one concurrency control protocol is able to achieve the two goals simultaneously.

Pessimistic locking and optimistic reading. The key insight behind PLOR is a novel hybridization of OCC and WOUND_WAIT. As shown in Figure 2c, PLOR follows a standard 2PL protocol to acquire locks before actually accessing records (i.e., *pessimistic locking*), but a transaction is allowed to ignore lock conflicts and access records without being blocked (i.e., *optimistic reading*). The benefits of such a design are two-fold. First, *optimistic reading* improves throughput by avoiding detecting conflicts in the read phase. As shown in Figure 2c, PLOR does not necessarily block $T2$ when $T1$ is holding the write lock, instead, $T2$ can read the record directly by ignoring the write lock, and safely commit the transaction before $T1$ commits. This is different from 2PL where reads are blocked by a writer all the way. Second, *pessimistic locking* requires transactions to keep their timestamps in the lock, enabling PLOR to commit transactions with the timestamp order, thus delivering low tail latency.

While the idea of PLOR is simple, two challenges still need to be tackled properly. **First**, by ignoring conflicts in the read phase, a transaction (e.g., $T1$) in PLOR may read incomplete records that are been written by another writer; moreover, when $T1$ commits, the records it reads may have already been modified by other writers, then $T1$ has to abort. If $T1$ owns a smaller timestamp than the conflicting transaction, it's impossible for PLOR to enforce the commit priority with the timestamp order. **Second**, OCC only locks for write operations, while PLOR requires both read and write locks. Hence, we require that the lock primitive in PLOR should cause minimal overhead. To address these challenges, PLOR introduces two technical contributions.

Enforce the commit priority via *delayed conflict detection*. To commit transactions with the timestamp order, PLOR ensures that ① conflicts can be *safely* ignored in the read phase, where transactions never read incomplete or uncommitted data (safety), and ② transactions are serialized with the timestamp order by resolving conflict in the commit phase (priority).

PLOR uses different ways to safely ignore read-write and write-write conflicts. To avoid detecting read-write conflicts in the read phase, PLOR introduce a private buffer for each working thread, as done in OCC. In this way, write operations always buffers modifications locally, preventing readers from reading incomplete or uncommitted data when they ignore write locks. Similarly, readers do not necessarily block writes since readers do not modify records anyway. For write-write conflicts, we delay acquiring write locks directly: for blind writes, which update a record without reading it beforehand, a transaction only needs to acquire write locks at the commit phase (see $T4$ in Figure 2c); for read-modify-writes, a transaction acquires shared locks in the read phase, and upgrades them to exclusive mode at the commit phase (e.g., $T3$). Note that delaying write locks is not always beneficial under certain workloads, so we keep this as an optional technique (§6.4).

PLOR performs conflict detection by delaying it to the commit phase (for priority). We use WOUND_WAIT to resolve potential conflicts leveraging the timestamps in the locks acquired in the read phase. In Figure 2c, $T1$ has to wait for $T2'$ to commit before it commits, since $T1$ owns a bigger timestamp; otherwise, $T1$ aborts the reader if $T1$'s timestamp is smaller. In this way, PLOR ensures that transactions with older timestamps are always committed first.

Minimizing the locking overhead with a *latch-free locker*. Traditionally, a lock primitive should maintain multiple data structures (as in WOUND_WAIT), which makes it hard to acquire/release locks atomically using lock-free data structures. However, this is not the case in PLOR – it acquires locks and detects conflicts in different phases. Understandably, ensuring the atomicity of the whole (un)locking process is hard, but is easy for every single phase. We will describe our *latch-free locker* in §4.2. For read-only transactions, PLOR further adopts a dynamic approach to consider both throughput and tail latency (see $T5$ in Figure 2c and §4.1).

Limitations. First, this paper mainly focuses on the impact of *request conflicts* on tail latencies. When handling requests with uneven distribution or variable sizes, existing task schedulers often suffer from head-of-line blocking, exacerbating the tail latency problem. Since these factors have been well studied in recent work [13, 37], we do not discuss them again. Second, we only consider single-node transactions and explore how PLOR performs within interactive services. we believe PLOR can be applied in distributed transactions as well, which we leave for future work. Third, PLOR optimizes tail latencies by compromising non-tail latencies (i.e., median latency). This is easy to understand: by granting higher priority on aborted transactions, newly invoked transactions are more likely to abort when they conflict. As a result, PLOR often exhibits higher latency in the non-tail part. However, user-facing service level objectives (SLOs) are typically determined by tail latencies. As our experiments will show in §6, non-tail latencies of PLOR are often in the range of several to hundreds of microseconds, which is orders of magnitude lower than that of 999p tail latencies.

4 DESIGN OF PLOR

In this section, we first describe PLOR (§4.1). Next, we describe the lock design (§4.2) and the correctness proof of PLOR (§4.3).

4.1 The PLOR Concurrency Control

We first describe a baseline of PLOR that only ignores read-write conflicts in the read phase. Ignoring write-write conflicts is considered as an optional optimization and will be described in §4.1.4. For simplicity, we first discuss how we run transactions that only contain reads and updates to existing keys. Inserts and range queries will be discussed later.

4.1.1 ***Data Structures.*** PLOR launches multiple worker threads to run transactions posted by clients. Each worker thread owns a *context* (`ctx`), which contains the following information when it runs a transaction.

- `ctx.wid` (16-bit, non-zero) is the ID of each worker thread.
- `ctx.ts` (47-bit) is the timestamp of the current running transaction, which determines the serial order at the commit phase.
- `ctx.status` (1-bit) tracks the state of each worker thread, which can be either `running` or `aborted`. With the WOUND_WAIT protocol, a worker thread kills a conflicting transaction by toggling this field of the thread running that transaction.

Note that, `wid`, `ts`, and `status` together form a 64-bit integer, which uniquely identifies the current running transaction. The contexts of all worker threads form an array, namely `ctx_arr[]`, indexed by the worker ID, which is globally accessible.

```
Data: ctx_arr[], read-set and write-set.
1  EXECUTE (wid):
2      ctx_arr[wid].ts = new_ts();
3      ctx_arr[wid].status = running;
4      ...
5      record_i.LOCKRD(ctx_arr[wid]);
6      ...
7      record_j.LOCKWR(ctx_arr[wid]);
8      COMMIT(wid);
```

Figure 3: A Transaction's Lifecycle in PLOR.

```
   w // ctx of the current writer that owns the lock.
   𝕎 // list of waiting writers in ascending T/S order.
   ℝ // list of readers in arriving time order.
1  LOCKRD (ctx):
2      ℝ.insert(ctx) ;                    // add myself to ℝ.
       // ℝ[...,ctx]: returns all elements between head and
          ctx; excl_sig: exclusive mode (predefined 64-bit).
3      while ℝ[...,ctx].contains(excl_sig) do
4          if ctx.ts < ctx_arr[w.wid].ts then
5              ctx_arr[w.wid].status ← aborted;
6          POLLONCE(ctx.wid);
7  LOCKWR (ctx):
8      𝕎.insert(ctx) ;                    // add myself to 𝕎.
       // update w from 0 to ctx via an atomic CAS.
9      if !CAS((w == 0) ? w ← ctx) then
           // CAS fails, wait till w = ctx or abort.
10         while w ≠ ctx do
               // w is younger, kill w.
11             if ctx.ts < w.ts then
12                 ctx_arr[w.wid].status ← aborted;
13             POLLONCE(ctx.wid);
14 UNLOCKRD (ctx):
15     ℝ.remove(ctx);
16 UNLOCKWR (ctx):
17     𝕎.remove(ctx);
18     if w == ctx then
19         ℝ.remove(excl_sig);
           // turn the lock over to the oldest waiter.
20         w ← 𝕎.first() ;                // w = 0 if 𝕎 is empty.
```

Figure 4: Lock Procedure.

Each worker thread also maintains read and write sets for the current transaction. *Read-set* contains pointers that point to records read by the transaction and *write-set* tracks modified records. The records that are both read and modified appear in both the *read-set* and *write-set*. Each item in the *write-set* also owns a *private buffer* to accommodate a transaction's updates to this record.

4.1.2 ***Read Phase.*** PLOR follows a conventional 2PL protocol to execute a transaction. As shown in Figure 3, a worker thread initializes its context before running a new transaction and always acquires the corresponding lock before actually accessing a record.

Figure 4 describes how a lock manager handles shared and exclusive lock requests, which exhibits the central idea of PLOR. As done in recent in-memory databases, we assign each record a lock manager for higher concurrency [44, 54]. As shown at the beginning of Figure 4, a lock manager keeps the following information: first, the current writer (w) that owners the lock exclusively; second, a waiting list for writers ($\mathbb{W}$) ordered in ascending timestamp order; and third, a list for readers ordered in arriving time order ($\mathbb{R}$).

To acquire a read lock, the worker thread simply ignores the current writer and insert itself directly to $\mathbb{R}$ (Line 2). When a writer

```
Data: ctx_arr[], read-set and write-set.
1  COMMIT (wid):
       // Phase 1: detect read-write conficts.
2      for wr in write-set do
3          //wr.LOCKWR(wid);          // when DWA is enabled.
4          wr.ℝ.insert(excl_sig);     // upgrade the lock.
           // ℝ[...,excl_sig): returns all elements between head
              and excl_sig (but w/o excl_sig).
5          for r in wr.ℝ[...,excl_sig) do
6              if ctx_arr[wid].ts < r.ts then
                   // kill r (younger than me).
7                  ctx_arr[r.wid].status ← aborted;
8              else
9                  while wr.ℝ.contains(r) do
10                     POLLONCE(wid) ;    // wait for it.
       // Phase 2 (commit point): release read locks.
11     for rd in read-set do
12         rd.UNLOCKRD(ctx_arr[wid]);
       // Phase 3: commit and release write locks.
13     for wr in write-set do
14         wr.commit_data(); // private_buf → DB.
15         wr.UNLOCKWR(ctx_arr[wid]);
16 POLLONCE (wid):
17     if ctx_arr[wid].status == aborted then
18         abort() ;          // check my status, abort if killed.
```

Figure 5: Commit Protocol.

(i.e., w) is committing, the reader should be blocked to avoid reading incomplete data (Lines 3 - 6), which will be described later. When acquiring a write lock, PLOR checks write-write conflicts but ignores read-write conflicts. To achieve this, it first adds itself to $\mathbb{W}$, and then grabs the lock by racing on the w variable using an atomic CAS operation. We use WOUND_WAIT to resolve write-write conflicts if a writer is holding the lock (i.e., $w \neq 0$, Line 9). If the requesting transaction has a smaller timestamp than the lock owner, it kills the owner by toggling its `status` to aborted (Line 12). Then, the lock requester waits until it acquires the lock (Lines 10 - 13). The atomicity of lock acquisition will be discussed in §4.2. Once the lock has been acquired, the worker accesses records and run the transaction logic. Note that all of a transaction's modifications are buffered in the private buffer before the commit phase.

4.1.3 ***Commit Phase.*** On transaction completion, a worker attempts to commit the transaction with three steps (see Figure 5). In *Phase 1*, the worker thread detects read-write conflicts of all records in the transaction's *write-set*. It first upgrades the locks in the *write-set* to *exclusive* mode (Line 4). Exclusive lock blocks all later readers that attempt to acquire read locks, preventing them from reading incomplete data. When a reader observes that a lock is in exclusive mode, it aborts this committing transaction if the reader owns a smaller timestamp, then, the reader waits until the exclusive lock has been removed (Lines 3 - 6 in Figure 4). The implementation of the exclusive locking mode will be discussed in §4.2. Once the lock is in exclusive mode, the worker then detects conflicts by scanning the readers in the lock: it kills all younger readers that own bigger timestamps (Line 7), and waits for the older readers to release the lock (Lines 9 - 10).

After Phase 1, the worker can safely commit the transaction. In *Phase 2*, the worker releases all the read locks in its *read-set*, which is done by removing itself from the list of readers (Line 15 in Figure 4). In *Phase 3*, the worker commits the modified records to the database

and releases the write locks. Releasing write locks consists of three steps (Lines 17 - 20 in Figure 4): remove the committed transaction from $\mathbb{W}$, disable the exclusive locking mode, and find the oldest waiter in $\mathbb{W}$ and turn the lock ownership over to it.

Liveness. As shown in the lifecycle of a transaction in Plor, a transaction can be killed by other conflicting ones at any stage when it holds some conflicting locks. Hence, a transaction needs to abort itself proactively if it has been killed, which avoids dependency cycles and thus ensures liveness. We achieve this by calling the PollOnce() function whenever a transaction waits for a lock synchronously (Lines 6 and 13 in Figure 4, Line 10 in Figure 5); Note that a committing transaction does not need to check its `status` anymore once it has finished phase 1; at this stage, it neither acquires more locks nor waits for lock dependencies. However, other transactions are unaware of such information and may still kill it (Line 7 in Figure 4). When the worker thread runs the next transaction, it will finally see this aborted status and abort the current transaction, causing unnecessary aborts. We address this issue by placing `status` and `ts` in the same 64-bit word (i.e., `ctx`). A thread aborts or activates a transaction by atomically changing the whole word, which succeeds only if the target thread is still using the original timestamp. The preemptive abort property of Plor (i.e., a lower-timestamp transaction can abort higher-timestamp conflicting transactions) ensures that the oldest transaction can always be committed, since other conflicting ones cannot block or abort it. Besides, the newly generated timestamp is monotonic, and an aborted transaction still uses its original timestamp when it reruns. Hence, an aborted transaction will finally become the oldest one after a certain number of retries and then commit, which guarantees the liveness of the protocol. Also, given that Plor is starvation-free and the number of worker threads is less than that of physical CPU cores, Plor will not deadlock the entire system even if we use busy-waits in the lock manager.

Insert operations. Conflict detection of *insert operations* is quite different since conflicts happen to non-existent records and there is nothing to lock. We address this issue by adopting the approach used in Silo [44], which inserts a new record in advance for the *insert* request during the read phase. An insert operation on key k is processed as follows: if k already exists, then the transaction is aborted; otherwise, the worker initializes a new record r and acquires the write lock of it in advance. Lock acquisition can always succeed since the record is still invisible to other concurrent transactions. Then the record is published to the database by inserting a mapping from $k \rightarrow r$ in the index structure. If the transaction aborts, the newly inserted mappings and records are removed.

Read-only transactions. Plor adopts a dynamic approach to consider both throughput and tail latency. At first, Plor runs read-only transactions via validation, as done in Silo. Read locks are used only after a transaction aborts many times (3 in our implementation). MVCC runs read-only transactions that never abort (e.g., Cicada [25]). However, MVCC introduces extra overhead for maintaining multiple versions and cleaning obsolete records; optimizing the performance of MVCC is out of the scope of this paper.

4.1.4 ***Delayed Write-Lock Acquisition.*** This part describes how we ignore write-write conflicts in the read phase by introducing *delayed write-lock acquisition (DWA)*. A *blind-write* record (i.e., a transaction writes a record without reading it beforehand) can be modified by other transactions arbitrarily before it is committed. Hence, we simply acquire their write locks at the commit phase (Line 3 in Figure 5). *Read-modify-write* records appear in both *read-set* and *write-set*. Therefore, we only acquire their read locks during the read phase, and upgrade them to *exclusive* mode before committing the transaction. *Delayed write-lock acquisition* further brings the following optimization opportunity: by acquiring write locks at the commit phase, a transaction already has a full *write-set*, so we can sort the *write-set* and acquire write locks in a deterministic global order to avoid deadlocks. By enabling *DWA*, Plor can finish the read phase without any lock blocking (similar to OCC).

However, we find that adding too much optimism in the read phase is not always beneficial. In stored-procedures, for instance, concurrent transactions reach the commit phase too quickly, and Plor often causes high abort rates when detecting conflicts using WOUND_WAIT. We will analyze this in detail in §6.4.

4.2 Latch-Free Locker

We introduce the *latch-free locker* to handle read-write and write-write conflicts in a lock-free manner. Plor decomposes lock acquisition and conflict detection into different phases, and we find this unique property greatly simplifies the way of implementing a lock primitive using lock-free data structures.

For read locks, we only need to ensure that the list of readers (i.e., $\mathbb{R}$) is atomic when multiple readers insert entries in $\mathbb{R}$ simultaneously. Luckily, there are many available open-sourced lock-free concurrent lists [3, 16]. Besides, detecting read-write conflicts should also be done atomically (the green dashed boxes in Figure 4 and 5). We need to ensure that once the lock is upgraded to an *exclusive* mode, all later readers are blocked, and all the existing readers are visible to the committing transaction. To realize this semantic, Plor introduces a pre-defined 64-bit value (i.e., *exlc_sig*) in the list of readers in a lock to indicate the exclusive mode. Specifically, $\mathbb{R}$ is ordered by the arriving time of each insertion; a committing transaction upgrades the lock by appending an *excl_sig* at the end of $\mathbb{R}$ (Line 4 in Figure 5), and then scans the readers whose entries appear before the *excl_sig* entry to detect read-write conflicts. When a transaction acquires a read lock, it appends an entry into $\mathbb{R}$ and then scans backwards to find if there is an *excl_sig* entry and block if necessary (Lines 3 - 6 in Figure 4).

For write locks, since we only allow one worker to exclusively own the write lock, this can be achieved via CAS instructions on an atomic word w (Line 9 in Figure 4). Detecting write-write conflicts needs to manipulate w and $\mathbb{W}$ simultaneously (red dashed boxes in Figure 4). However, pushing an item in $\mathbb{W}$ and changing the value of w cannot be performed atomically; similarly, turning the lock over to the oldest waiter when releasing the lock (Line 20 in Figure 4) cannot be done atomically either. One inconsistent case arises when a lock requester (T_i) sees a non-zero w, and at the same time, the lock owner releases the lock, and grants the lock to the oldest waiter in $\mathbb{W}$ before T_i is visible in the list. Hence, T_i might wait for someone that owns a higher timestamp. To address this, Plor requires the waiters in $\mathbb{W}$ to compare its timestamp with w repeatedly and kill the lock owner if such inconsistent cases occur (Lines 10 - 13 in Figure 4).

Lock-free list using an atomic word. We find that implementing a lock-free list can be further simplified by manipulating the bits within an 8-byte atomic word. Specifically, we assign each worker thread a bit in the atomic word, and the offset of the bit is determined by the worker thread's ID (i.e., `wid`). To insert an item into the list, the worker thread simply sets the corresponding bit to 1 via a `fecth_and_add`. To support *exclusive mode*, the last bit of each atomic word (i.e., $\mathbb{R}$) is reserved to act as an *excl_sig* entry. When a worker acquires a read lock, it uses `fetch_and_add` to set the corresponding bit to 1, and checks whether the *excl_sig* entry has already been set. If so, the reader clears its bit to zero and wait accordingly. As a result, an 8-byte atomic word can support at most 63 worker threads, which is enough on our platform. We can still use lock-free queues when more worker threads are added.

4.3 Correctness Proof of Plor

In this section, we prove that Plor is able to correctly enforce serializability. To show this, we will prove that Plor guarantees *conflict serializability.*

4.3.1 Preliminaries. We first define several frequently used notations. A *transaction* T_i is a sequence of operations, where each operation can be a read $r_i(X)$, write $w_i(X)$, lock $l_i(X)$, or unlock $u_i(X)$ on record X. It will be helpful for our purposes to distinguish the type of a lock by writing $l^r(X)$, $l^w(X)$, or $l^{w+e}(X)$ for read lock, write lock, and exclusive write lock respectively. Similarly, we will write $u^r(X)$ and $u^w(X)$ for unlocks.

Definition 1 (Schedule). *A* schedule *S for transactions $T_1, T_2, \ldots, T_n$ is any interleaving of the operations in the transactions such that the order of operations in the same transaction is maintained.*

Definition 2 (Serial Schedule). *A schedule is* serial *if no transaction starts until another transaction has ended.*

We next formally define conflicts. A *write-write (WW) conflict* in a schedule S is defined as a pair $(w_i(X), w_j(X))$ such that $i \neq j$ and $w_i(X)$ occurs before $w_j(X)$ in S. We can similarly define read-write (RW) conflicts and write-read (WR) conflicts. A *conflict* is any WW, WR, or RW conflict.

We can now define two important notions.

Definition 3 (Conflict Equivalence). *Two schedules S, S' on the same transactions are* conflict equivalent *if they have exactly the same set of conflicts.*

Definition 4 (Conflict Serializable). *A schedule is* conflict serializable *if it is conflict equivalent to a serial schedule.*

From here on, let S be a schedule produced by Plor for a set of transactions $T_1, \ldots, T_n$ that have committed. S has the following properties, which will be of use in the proof:

A. At any point in the schedule, at most one transaction can hold a write lock for the same record X. This follows from the atomic execution of lines 9 - 13 in Figure 4.
B. All lock operations precede all the unlock operations within the same transaction. This property is similar to that of the standard 2PL protocol. Indeed, in Plor a transaction releases all the locks when it commits.
C. A transaction can write a record only after its lock has been upgraded to *exclusive* mode (Lines 4 and 14 in Figure 5), despite that *DWA* has been enabled or not, during which other readers and writers cannot access this record.

4.3.2 Main Proof.

Theorem 1. *Let S be any schedule produced by Plor. Then, S is conflict serializable.*

Proof. Similar to existing methods used to prove the conflict serializability of 2PL, we will use *induction* in the number of transactions in the schedule S.

For the base case of the induction, S contains only one transaction T. In this case, S is itself a serial schedule, and hence it is trivially conflict serializable.

For the induction step, let S be a schedule of n transactions $T_1, T_2, \ldots, T_n$. By the inductive hypothesis, any schedule consisting of at most $n-1$ transactions must be conflict serializable. We show that S is also conflict serializable.

Let $u_i(X)$ be the first unlock operation found in S, which is issued by transaction T_i.

$$S : \ldots, \ldots, u_i(X), \ldots, \ldots$$

Let S' be the schedule constructed by moving all the operations of T_i forward to the beginning of S while maintaining their order:

$$S' : \underbrace{\ldots, \ldots, \ldots, \ldots}_{\text{operations of } T_i}, \overbrace{\underbrace{\ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots, \ldots}_{\text{operations of other } n-1 \text{ transactions}}}^{S''}$$

Claim 1: *S' and S are conflict equivalent.*

Proof of the claim. We show that conflict equivalence holds for all three types of conflicts.

Write-write conflict. Consider a write operation $w_i(Y)$ by transaction T_i on record Y. We will show that any other write operation $w_j(Y)$ on the same record must happen after $w_i(Y)$. Indeed, suppose for the sake of contradiction that $w_j(Y)$ occurs before $w_i(Y)$ in the schedule:

$$S : \ldots, w_j(Y), \ldots, w_i(Y), \ldots$$

Before any write, in Plor the transaction must own the write lock for the corresponding record. By *Property A*, at most one transaction can hold this lock at any point, hence T_j must have released the lock before T_i acquires it in order to write Y. Hence, the schedule must be as follows:

$$S : \ldots, l_j^w(Y), \ldots, w_j(Y), \ldots, u_j^w(Y), \ldots, l_i^w(Y), \ldots, w_i(Y), \ldots$$

However, by our assumption, $u_i(X)$ is the first unlock operation in S, and thus must occur before $u_j^w(Y)$, indicating that $u_i(X)$ precedes $l_i(Y)$ in the schedule, which is a contradiction to *Property B.*

Read-write conflict. Consider a write operation $w_i(Y)$ of transaction T_i on record Y. We will show that any other read operation $r_j(Y)$ on the same record must happen after $w_i(Y)$ in S. Indeed, suppose for the sake of contradiction that $r_j(Y)$ occurs before $w_i(Y)$ in the schedule:

$$S : \ldots, r_j(Y), \ldots, w_i(Y), \ldots$$

Before the write $w_i(Y)$, the transaction T_i must acquire the write lock $l_i^w(Y)$ in exclusive mode (i.e., $l_i^{w+e}(Y)$). We argue that $l_i^{w+e}(Y)$ must appear after $r_j(Y)$ in the schedule. Indeed, if not, T_i would

have to release the lock on Y before $w_i(Y)$, a contradiction. Hence, the schedule must be as follows:

$$S : \ldots, l_j^r(Y) \ldots, r_j(Y), \ldots, l_i^{w+e}(Y), \ldots, w_i(Y), \ldots$$

Now, consider the readers' list of record Y when T_i upgrades the write lock to exclusive mode. Since T_j is not aborted, this means that at some earlier point T_i must have released the read lock on Y. Hence, the schedule is:

$$S : \ldots, l_j^r(Y) \ldots, r_j(Y), \ldots, u_j^r(Y), \ldots, l_i^{w+e}(Y), \ldots, w_i(Y), \ldots$$

By our assumption, $u_i(X)$ is the first unlock operation in S, and thus must occur before $u_j(Y)$. But that means that $u_i(X)$ precedes $l_i^{w+e}(Y)$ in the schedule, which is a contradiction to *Property B*.

Write-read conflict. Consider a read operation $r_i(Y)$ by transaction T_i on record Y. We will show that any other write operation $w_j(Y)$ on the same record must happen after $r_i(Y)$ in S. Indeed, suppose for the sake of contradiction that $w_j(Y)$ occurs before $r_i(Y)$ in the schedule:

$$S : \ldots, w_j(Y), \ldots, r_i(Y), \ldots$$

Transaction T_j must acquire the write lock of Y and upgrade it to *exclusive mode* before writing. Furthermore, before reading transaction T_i must acquire a read lock $l_i^r(Y)$. We now distinguish two cases, depending on whether $l_i^r(Y)$ occurs before or after $l_j^{w+e}(Y)$.

In the first case, $l_i^r(Y)$ occurs before $l_j^{w+e}(Y)$. Then, the schedule looks as follows:

$$S : \ldots, l_i^r(Y), \ldots, l_j^{w+e}(Y), \ldots, w_j(Y), \ldots, r_i(Y), \ldots$$

Following a similar argument to the one in the read-write conflict case, T_i must release the read lock on Y before T_j upgrades the lock to exclusive mode. But then $u_i^r(Y)$ would occur before T_i reads Y, a contradiction to how Plor behaves.

In the second case, $l_i^r(Y)$ occurs after $l_j^{w+e}(Y)$, and hence the schedule is as follows:

$$S : \ldots, l_j^{w+e}(Y), \ldots, l_i^r(Y), \ldots, r_i(Y), \ldots$$

But then, because of lines 3 - 6 in Figure 4, T_j must unlock Y before T_i obtains the read lock. Hence, $u_j^w(Y)$ occurs before $l_i^r(Y)$ in the schedule. By our assumption, $u_i(X)$ is the first unlock operation in S, and thus must occur before $u_j^w(Y)$. But that means that $u_i(X)$ precedes $l_i^r(Y)$ in the schedule, contradicting *Property B*. □

To conclude, let us consider again the schedule S'. This schedule consists of all operations in transaction T_i, followed by a subschedule S'' of $n-1$ transactions. By the induction hypothesis, S'' is conflict equivalent to a serial schedule. Consequently, the initial schedule S' is also equivalent to a serial schedule. Since by Claim 1 S and S' are conflict equivalent, S is conflict serializable as well. □

5 IMPLEMENTATION

In our implementation, we apply Plor in DBx1000 [54], which is a multi-threaded, shared-everything OLTP DBMS. The tables in DBx1000 are stored in a row-oriented manner. Since DBx1000 only supports hash-based indexes, we revise it to use Masstree [26] as an alternative ordered index scheme.

Interactive Processing model. Most in-memory database prototypes in academia [19, 44, 49, 55] run transactions in stored-procedure mode, where all accesses in a transaction and the execution logic are ready before execution. A recent study finds that *interactive transaction processing* still dominates in production environments [32]. As such, we also revise DBx1000 to support interactive processing mode. Specifically, we decouple Plor into two parts, including a transaction processing engine (runs on clients) to execute the transaction logic, and a storage engine to manage the actual data. Clients interact with the storage engine by sending data queries via networked RPCs. In our implementation, we use eRPC [21] to send requests, which is a fast and general-purpose remote procedure call library.

Data Durability. Many recent DBMSs [25, 44, 55] ensure data durability via parallel logging. The basic idea is to group transactions into epochs and perform logging in batches. This optimization is based on the fact that accessing external storage devices is extremely slow. While parallel logging improves throughput, it impacts latency inevitably by delaying the responses to clients. Fortunately, emerging non-volatile memory technologies, such as Intel's Optane DC Persistent Memory (Optane DCPMM) [28], exhibit extremely low write latencies (around 100 *ns*) and provide us with the opportunity to log data immediately without group commit and separate loggers. To understand how logging to Optane DCPMMs impacts tail latency and throughput, we implement both *redo* and *undo* logging in Plor. For *redo* logging, a transaction buffers the updated records in DRAM, and never makes updates to the database during the read phase. At the commit phase, updates are appended to the log file before committing them in place. Hence, a transaction performs *redo* logging only when it can commit. For *undo* logging, the old version of a record is logged immediately before modifying it. *Undo* logging makes aborts more expensive due to the extra logging overhead in the read phase.

6 EVALUATION

In our evaluation, we try to answer the following questions:

- Does Plor achieve the goal of delivering low tail latency and high throughput under high-contention workloads on various setups?
- Does Plor still provide comparable throughput to that of Silo under low contention workloads?
- How does each technique in Plor help with optimizing tail latency and throughput?

6.1 Experimental Setup

Testbed. Our experiments run on a machine with two Intel® Xeon® Gold 6240M CPUs (each with 18 physical cores and 25 MiB LLC, clocked at 2.6 GHz), 192 GiB DDR4 DRAM, and 4 Optane DCPMMs (256 GiB per module, 1 TB in total). The server is installed with Ubuntu 18.04 and Linux 4.15. To get steady results, the experiments pin threads to different cores, which are evenly distributed across two NUMA nodes.

By default, transactions are processed in stored-procedure mode and persistent logging is disabled. When evaluating the effect of persistent logging, the worker threads log data to local Optane DIMMs that are attached to the same CPU. Optane DCPMMs are

configured in App-Direct mode and their spaces are managed by the NOVA file system [52], which is a scalable NVM-aware file system. When evaluating the *interactive transaction processing* mode, the experiments run on two such machines. They are connected with a Mellanox MSB7790-ES2F switch using MCX555A-ECAT ConnectX-5 EDR HCAs, which support the 100Gbps Infiniband network.

Compared Algorithms. DBx1000 includes a pluggable lock manager that supports different concurrency control schemes, and this allows us to compare them within the same system. We choose three 2PL-based schemes (NO_WAIT, WAIT_DIE and WOUND_WAIT), two OCC schemes (Silo [44] and TicToc [55]) and one hybrid scheme (MOCC [49]) for comparison in this paper. MOCC [49] uses NO_WAIT and OCC dynamically to access records based on their hotness (we differentiate the design details of MOCC and Plor in §7). We believe that they are representative of the types of concurrency control protocols we target. Other approaches are possible, such as deterministic [43] or static analysis [50, 51] approaches. However, these types of approaches, unlike ours, require that transactions' read and write sets to be known a priori. For this reason, we also disable the *retrospective lock list* (RLL) technique in MOCC, since RLL assumes that transactions are deterministic (the read and write sets of an aborted transaction never change when it reruns).

Workload. Experiments use two standard benchmarks, including YCSB [9] and TPC-C[1]. TPC-C models online transaction processing (OLTP) databases with a configurable number of *warehouses*. It consists of nine tables, where customers are assigned to a set of districts within a local warehouse, and orders are placed in those districts. Among the five types of transactions, `Payment` and `NewOrder` make up 88% of the default TPC-C mix. They mostly interact with its local warehouses, but 10% of `NewOrder` and 15% of `Payment` transactions access a remote warehouse. `Stock-Level` represents a heavy read-only database transaction and has relaxed isolation requirements (i.e., read-committed). To execute such transactions, 2PL releases the lock immediately after accessing a new record, and OCC skips the validation phase directly. DBx1000 uses hash indexes for the tables that do not require range queries, and we still keep this optimization.

The Yahoo! Cloud Serving Benchmark (YCSB) emulates large-scale on-line services. By default, each query accesses a single record (1KB) and the contention level is controlled via a Zipfian distribution by tuning the parameter θ. YCSB-A and YCSB-B are used in this paper. YCSB-A represents the update-heavy and high-contention workload (50% reads and 50% writes, parameter=0.99). YCSB-B represents the read-intensive workload (95% reads and 5% writes, parameter=0.5). We slightly modify the two workloads to use a bimodal distribution of transaction sizes, where 90% are small transactions (4 operations per transaction) while the rest 10% are big ones (16 operations per transaction).

Measurements. When we run transactions in stored-procedure mode, we generate transaction requests locally, instead of receiving requests from remote clients through the network, thus preventing irrelevant factors (e.g., network, queuing delays) from impacting tail latencies. When collecting latencies, each transaction request is timestamped with a start time when it is invoked by the worker thread. In the context of interactive mode, the start timestamp is generated by the transaction processing engine. The end-to-end latency is computed only after the transaction has been committed.

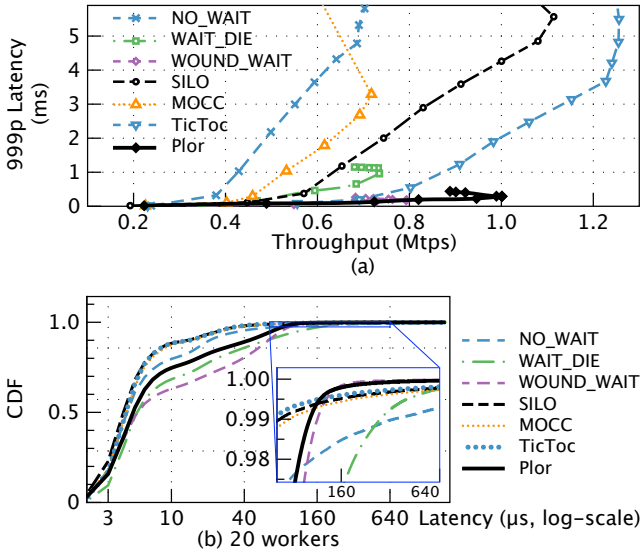

Figure 6: YCSB-A (θ = 0.99, w:50%, r:50%). ***Stored-procedure; (a) the 99.9th percentile latency against throughput by varying the number of workers; (b) the latency distribution when running 20 workers.***

6.2 High-Contention Workloads

We first evaluate Plor under high contention workloads. Both stored-procedure and interactive processing modes are evaluated. In the stored-procedure mode, we disable *deferred write-lock acquisition* by default, and we will analyze it in §6.4 in detail.

6.2.1 ***Stored-Procedure Mode.*** We first run high-contention workloads in stored-procedure mode.

YCSB-A. Figure 6 shows the 99.9th percentile latency (i.e., 999p) against the throughput as we increase the number of worker threads (from 1, 4, 8, to 36 with a step of 4). For Silo and TicToc, we add up to 48 threads to get their peak throughput. The latency distribution when we run 20 threads are shown at the bottom of Figure 6. We make the following observations.

First, Plor achieves 25% to 42% higher peak throughput than that of 2PL schemes, and only underperforms OCC-based approaches by 9% (for Silo) and 19% (for TicToc). Plor performs better than 2PL due to the following reasons. 1) Like OCC, Plor removes read-write blocking overhead during the read phase, and only detect conflicts at the commit phase, which improves concurrency. 2) Plor introduces the latch-free locker, which avoids the locking overhead (we will analyze this in §6.4 in detail). TicToc delivers the highest peak throughput since it lazily computes a valid commit timestamp based on the records accessed by the transaction, which permits more concurrency. Note that Silo and TicToc require more than 40 threads to reach their maximum throughput, while Plor only requires 20 threads. The main reason lies in that Silo and TicToc can only decide whether or not to abort a transaction at the commit phase, which wastes CPU cycles under high abort rates. Transactions running with Plor, instead, can be killed at either the read phase or the commit phase. After reaching the peak point, Plor's performance reduces slightly by 10% as the number of threads increases, and this also happens in WAIT_DIE and WOUND_WAIT. We believe this can be addressed by using better admission control policies.

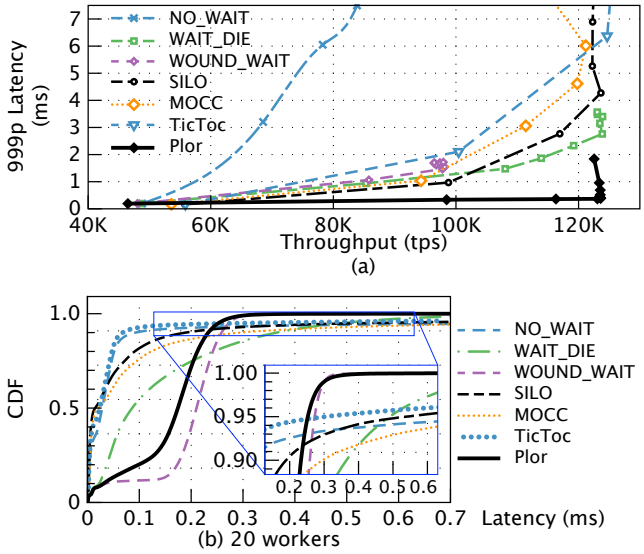

Figure 7: TPC-C with 1 warehouse. ***Stored-procedure; (a) the 99.9th percentile latency against throughput by varying the number of workers; (b) the latency distribution when running 20 workers.***

Second, Plor shows comparable 99.9th percentile latencies to that of WOUND_WAIT, which are significantly lower than that of Silo and TicToc. As shown in Figure 6a, for a target throughput running at 1 million tps, Plor is able to restrict its 999p latency within 294 μs, which is 14.5× and 8.8× lower than that of Silo and TicToc, respectively. Plor exhibits low tail latency since it still uses WOUND_WAIT to resolve conflicts, which commits transactions with the timestamp order. The tail latency and throughput of MOCC are between those of NO_WAIT and Silo, since it is a combination of the two algorithms.

Third, the *non-tail latencies* (i.e., from 0th to Nth percentile, where $N = 0.995$ in Figure 6b) of Plor are higher than that of Silo. This is as expected since Silo and Plor have close peak throughput, so they should also exhibit similar average latencies. Given that Plor shows lower tail latency, it must have higher *non-tail* latencies.

TPC-C. We set the number of warehouses to 1 to model the high-contention workload and the results are shown in Figure 7. The evaluated systems perform differently in two aspects compared to that with the YCSB-A workload. First, we observe that Plor's tail latency is lower than that of Silo and TicToc in the range of 92th to 100th percentile (see Figure 7b), which is different from that in YCSB-A (99.5th to 100th). We specify two main reasons. First, TPC-C has a higher contention level when using a single warehouse. In such a setting, all the `Payment` and `NewOrder` transactions need to access the same warehouse and thus always conflict with each other. Second, TPC-C consists of a complicated mixture of transactions with different execution logic, where some of them only access a few records (e.g., `Payment`), while others may access tens to hundreds of records in a single transaction (e.g., `Delivery` and `Stock-Level`). We also find that long transactions are more likely to abort in Silo and TicToc, exacerbating the tail latency problem.

Second, WAIT_DIE and MOCC achieve almost the same peak throughput as that of Plor and Silo. By profiling the details, we observe that transactions in TPC-C often cause *single-point* conflicts

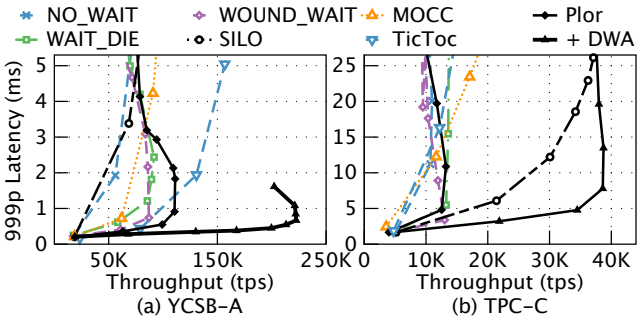

Figure 8: Interactive Transaction Processing. ***(a) and (b) show the 99.9th percentile latency against throughput for YCSB-A and TPC-C workloads, respectively, under interactive mode.***

on the `warehouse` record. In WAIT_DIE, such conflicts are handled better than that in WOUND_WAIT. For example, when a `NewOrder` transaction is holding the read lock of the warehouse, the following `Payment` transaction with a higher timestamp is aborted when it tries to acquire the write lock. If another `NewOrder` transaction comes, it still can acquire the read lock since it does not conflict with the former one. However, this won't happen in WOUND_WAIT, where the `Payment` transaction is placed in the waiting queue, which again blocks the following `NewOrder` transactions. In MOCC, only a few frequently updated records are locked (e.g., `warehouse`), and this helps MOCC to avoid unnecessary aborts while without introducing extra locking overhead on other records. The throughput of TicToc peaks at 134 Ktps (not shown in the figure since it has a 999p latency of more than 20ms at this point), which is 7.5% higher than that of Plor.

6.2.2 ***Interactive Processing Mode.*** We now analyze the performance of Plor running in the *interactive processing* mode. To prevent the data storage engine from becoming the bottleneck, two machines that run the two engines always use the same number of worker threads. *Delayed write-lock acquisition* is evaluated in this part (denoted as *+DWA*).

Figure 8a and 8b show the results with YCSB-A and TPC-C workloads, respectively. Under the YCSB-A workload, Silo shows very poor performance. We observe it exhibits the highest abort ratio among the compared schemes, and these aborted transactions consume most CPU cycles to access remote records. TicToc performs better than Silo since it lazily generates commit timestamps, which incurs a fewer number of aborts. Plor outperforms WOUND_WAIT by 49% in terms of peak throughput and still keeps comparable 999p latencies. When the *delayed write-lock acquisition* technique is enabled, *+DWA* further improves throughput by 2×. *+DMA* has almost the same abort ratio as that of Silo. However, this does not impact its efficiency, since aborted transactions in *+DWA* can be killed in advance by conflicting transactions, and this avoids most of the unnecessary remote accesses. Under the TPC-C workload (see Figure 8b), the major difference is that Silo has almost the same peak throughput as that of *+DWA*. As illustrated before, TPC-C with one warehouse only permits very limited concurrency, and both Silo and *+DWA* reach the maximum performance that can be achieved. Even so, *+DWA* still achieves 4× lower 999p latency as their throughputs saturate. Their latency distributions exhibit almost the same trend as that in Figures 6 and 7. Due to the space limitation, we do not show them again.

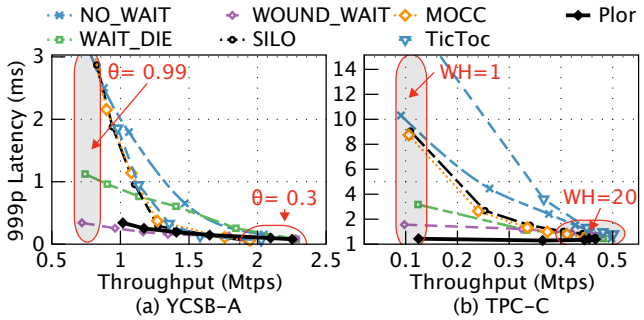

Figure 9: Performance with Varying Contention Levels. ***Stored-procedure, 20 threads; (a) and (b) show the 99.9th percentile latency against throughput for YCSB-A and TPC-C workloads, by changing the skewness or the number of warewhouses; WH indicates the number of warehouses.***

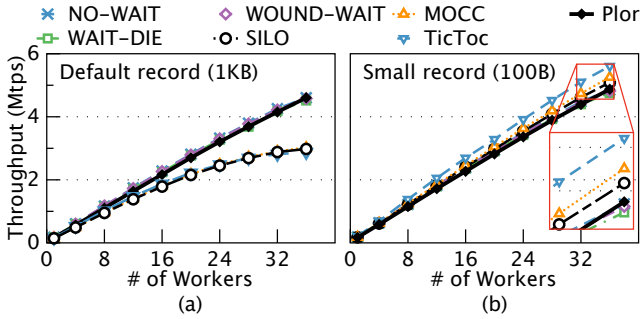

Figure 10: YCSB-B Throughput. ***Stored-procedure; (a) uses default record size (i.e., 1KB); (b) uses small record size (i.e., 10B).***

6.3 Varying-Contention Workloads

In this part, we evaluate the performance of Plor by varying the contention level. Specifically, when running the YCSB-A workload, we change the skewness by turning the θ parameter from 0.99 to 0.3; with the TPC-C workload, we increase the number of warehouses from 1 to 20. All experiments are run in stored-procedure with 20 threads (*DWA* in Plor is disabled). As shown in Figure 9, we can find that all evaluated systems exhibit lower tail latency and higher throughput as the contention level decreases. Among them, Plor consistently provides the lowest tail latency under the two workloads as the contention level varies. For the TPC-C workload, Plor's 999p latency is almost unchanged when more warehouses are added, while SILO and TicToc show a reduction of 17× and 36×, respectively, for the 999p latency.

We further evaluate the performance of low-contention workloads (i.e, YCSB-B) in stored-procedures. Figure 10 only reports throughputs since aborts are rare and the tail latency is almost unchanged. By using the default record size (i.e., 1 KB, in Figure 10a), we find that all 2PL-based schemes, including Plor, scale linearly. Two reasons account for their high performance. First, YCSB-B is read-dominated and conflicts are rare; Second, 2PL and Plor only need to copy data for write operations (from the private buffer to the database), whose overhead is almost negligible since YCSB-B only contains 5% write operations. Silo, TicToc and MOCC, instead, have to copy all the records into the private buffer for a transaction, and thus deliver the worst throughput. In Figure 10b, we modify the record size to 10 bytes to explore the maximum throughput

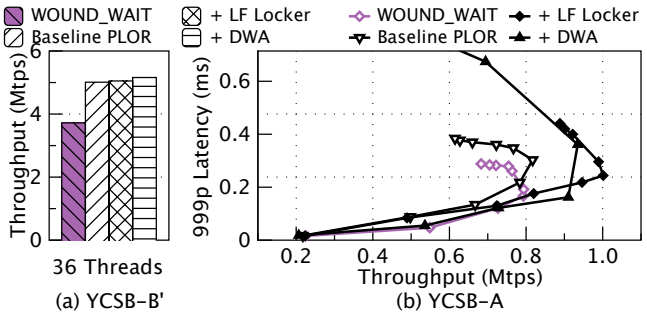

Figure 11: Analysis of Internal Factors. ***Stored-procedure; (a) the throughput of each mechanism under YCSB-B′, (b) the 999p latency against throughput under YCSB-A; LF Locker: latch-free locker.***

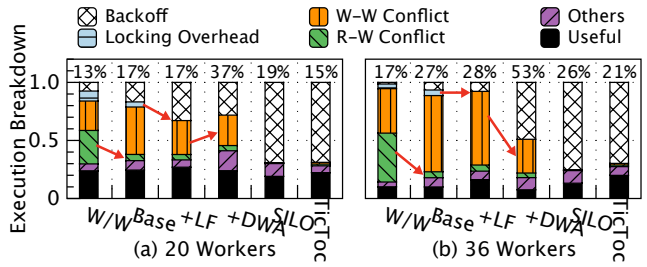

Figure 12: Execution time breakdown of internal factors. ***The percentage above each bar is the abort ratio, and the red arrows point out which metric is the most relevant to a certain mechanism.***

each concurrency control scheme can achieve. We observe that all schemes scale linearly as the number of threads increases. Moreover, the throughput of TicToc is slightly higher than the other schemes since it incurs lower locking overhead.

6.4 Factor Analysis

Internal Mechanisms. To understand in great detail the benefits and overhead brought by various techniques of Plor, we show a factor analysis in Figure 11. Among the evaluated techniques, *+LF Locker* adds the *latch-free locker* technique upon *Baseline Plor*, and *+DWA* enables the *delayed write-lock acquisition* in Plor. To highlight the effects of each technique, YCSB-B′ is used in 11a, which is a medium-contended workload with a parameter of 0.8. Stored-procedure mode is used in this part.

The major takeaway from Figure 11a is that *avoiding read-write conflict detection* is an important optimization for Plor when processing read-dominated workloads, which improves throughput by 35%. Both *+LF Locker* and *+DWA* show limited performance improvement. For *+LF Locker*, since YCSB-B′ is medium-contended, the lock thrashing overhead is not high. *+DWA* target at reducing write-write conflict, which is not the main overhead in YCSB-B′.

Figure 11b shows the 99.9th percentile latency as a function of throughput under the YCSB-A workload. To help with understanding their performance, the execution time breakdown of each mechanism is collected and shown in Figure 12. We make the following two observations from the figures:

First, *+LF Locker* improves throughput by 25% compared to that of *Base Plor*. By analyzing the execution breakdown, *+LF Locker* reduces the locking overhead from 4.4% to less than 0.1%. *Baseline Plor* shows almost the same peak throughput as that of

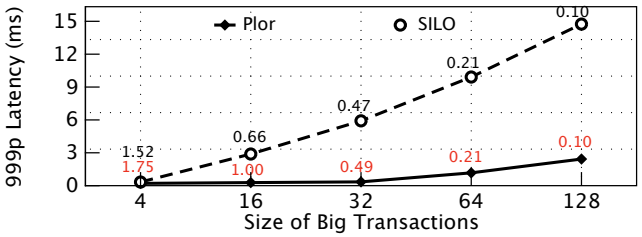

Figure 13: Effects of Request Distribution on Tail Latency. ***We use YCSB-A workload with 20 threads by varying the size of big transactions; the value on each point indicate the throughput (Mtps).***

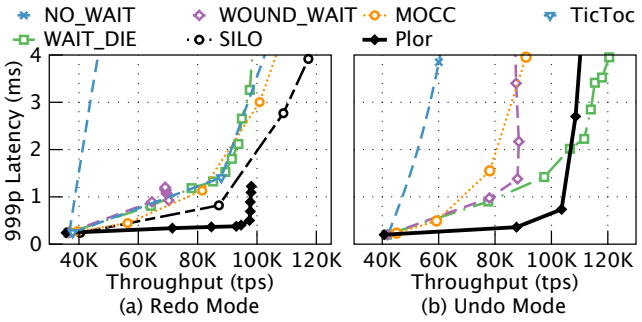

Figure 14: Effects of Persistent Logging. ***(a) and (b) show the 999p latency against throughput for redo and undo logging, respectively. We use TPC-C (1 warehouse) in stored-procedure mode.***

WOUND_WAIT; however, this does not indicate that the baseline does not have any benefits. By delaying the detection of conflicts, Plor separates the lock acquisition and conflict detection into different phases. Since *Baseline Plor* uses `mutex` locks to serialize concurrent accesses to the locking states, it has to acquire twice as many `mutex` locks as WOUND_WAIT, which eliminates the benefits Plor brings. Even so, *Baseline Plor* still reduces read-write conflicts from 29% to 5%.

Second, *+DWA* mitigates the write-write conflicts greatly (from 63% to 29% with 36 threads). However, *+DWA*'s peak throughput is slightly lower than that of *+LF Locker* and its performance collapses as the number of threads increases. From Figure 12b we can observe that *+DWA* shows a much higher abort ratio. During the read phase, *+DWA* processes transactions without any lock blocking (similar to OCC). In stored-procedure mode, transactions running with *+DWA* will quickly reach the commit phase. However, *+DWA* resolves conflicts with WOUND_WAIT, which makes most transactions execute too optimistically and abort in the end.

Effects of Request Distribution. As described in §6.1, we revise the YCSB-A workload to use a bimodal distribution of transaction sizes (i.e., 10% are big transactions and the rest 90% are small ones). In this part, we study how the request distribution hurts tail latencies by varying the size of big transactions. As shown in Figure 13b, the 999p latency of Silo increases by 45× as we change the size of big transactions from 4 to 128 (32× of growth). This indicates that OCC aggravates the tail latency problem when it meets workloads mixed with big transactions. Instead, the 999p latency of Plor increases only by 12×, which is even slower than the growth of the transaction sizes. This further confirms that abort times is the dominating factor that hurts tail latencies in Plor for high-contention workloads.

Effects of Persistent Logging. Figure 14 evaluates the effects of *redo* and *undo* logging on various concurrency control schemes. We still use the TPC-C workload in stored-procedure mode. In general, logging in Optane DCPMMs adds very limited overhead to each scheme. By comparing the results with that in Figure 7, we find that the throughput of Silo with *redo* logging is almost unchanged. Optane DCPMMs exhibits comparable write latency to that of DRAM, so the logging overhead is not significant. Besides, with *redo* logging, a transaction logs data only after it reaches the commit point, so it never causes unnecessary logging overhead. We do not evaluate Silo, TicToc and MOCC with the *undo* logging mode since they never perform in-place updates before the commit phase – *undo* logging mode requires an update to be logged before it is modified in-place. 2PL schemes perform better under *undo* mode, where they only have a performance decline within 10%. 2PL can abort a transaction at any phase, so they do not cause unnecessary logging overhead too much when aborting a transaction. *Undo* logging causes higher tail latency than *redo*, but still, Plor exhibits the lowest tail latency among the compared systems. 2PL schemes are less efficient under *redo* mode. The reason is that 2PL locks the records all the way, and logging further prolongs the lock holding time, limiting concurrency.

Effects of Commit Priority. Real-time database systems (RTDBS), an important line of research, use timing constraints to yield reliable responses [18, 41, 45]. They achieve this by granting a deadline to each transaction and trying to minimize the percentage of late transactions. When resolving conflicts, RTDBS uses deadline as the commit priority between conflicting transactions, i.e., transactions with earlier deadlines have a higher priority to commit.

In this part, we apply the deadline-based commit priority policy in Plor and compare its performance against our original version, which uses arrival timestamp as the commit priority. The deadline assignment formula used here is similar to [18], i.e., $D_T = A_T + SF * R_T$, where D_T, A_T, and R_T are the deadline, arrival time and resource time, respectively, of the running transaction, while SF is a slack factor. For simplicity, the resource time is equal to the number of records accessed by the current transaction. We run the experiment with different values of SF and the results are shown in Figure 15. We can find that as the value of SF increases, the tail latency of Plor increases as well. As we analyzed before, big transactions often incur higher tail latency. However, from the formula we can understand that, big transactions typically have later deadlines when $SF > 0$, which indicates that big transactions have lower commit priority and they are more likely to abort in the first few attempts to commit, leading to the high tail latency.

7 RELATED WORKS

Tail Latency Optimization. A number of recent works achieve microsecond-scale SLOs (service-level objectives) in different layers in the operating system. For example, *dataplane operating systems* optimize for throughput and tail latency by separating the OS dataplane from the OS control plane [5, 15, 34, 35]; *microsecond-scale core scheduling systems* introduce user-space thread allocation and scheduling policies [20, 29, 31, 36, 47]; *microsecond-scale queue manager* controls tail latency by changing the amount of resources

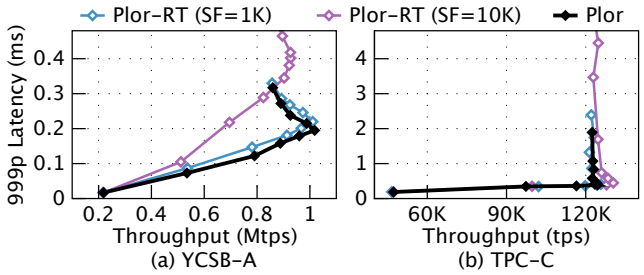

Figure 15: Effects of Commit Priority. ***Stored-procedures, Plor-RT uses deadline as the commit priority. (a) and (b) show the 999p latency against throughput with different value of SF (slack factor) under YCSB-A and TPC-C (1 warehouse) workloads.***

dedicated to long-running requests [13, 17, 24, 37]; and *storage management* reduces the impact that caused by background tasks or hardware resources [4, 6]. Plor starts from a different perspective by studying how high-contention workloads impact the tail latency in transactional systems.

Transaction Scheduling. Ding et al. [14] present a framework to batch across a transaction's entire life cycle for OLTP systems based on OCC. They show that batching does not necessarily indicate high tail latency, instead, by adopting proper reordering policies, batching even has the chance of reducing tail latency. Different from them, Plor does not reorder or batch transactions. Deterministic databases (e.g., Calvin [43], SLOG [38], etc.) order transactions' conflicting record accesses deterministically before executing them, where transactions never abort except for outside events. Dast is a recent variant of deterministic databases designed to achieve low tail latency for serializable transactions in edge computing. It reorders cross-region transactions to prevent them from blocking intra-region ones. However, deterministic databases require a priori knowledge of read/write sets of transactions, which is not always the case for some general transaction workloads [1, 2]; besides, some of them also do not scale well on multi-core platforms [50]. Plor is designed to run general transactions while delivering both high throughput and low tail latency.

Optimizing High-Contention Workloads. Transaction chopping [40, 57] and its recent followers (e.g., IC3[50] and Runtime Pipelining [27, 51]) decompose a transaction into a series of atomic sub-transactions and allow them to run in parallel. Transaction chopping improves throughput under high-contention workloads, but it is not a direct solution for reducing the tail latency. For instance, IC3 executes sub-transactions optimistically, which cannot prevent aborted transactions from aborting again. Besides, similar to deterministic databases, these approaches also require the full knowledge of read and write sets prior to transaction execution.

Multi-version concurrency control (MVCC) and timestamp ordering are another line of research that optimize high-contention workloads. In MVCC (e.g., MV2PL [7], Cicada [25]), each record has multiple versions, where locks acquired for reading data do not conflict with locks acquired for writing data. Similar optimization also exists in OCC-based approaches (e.g., Silo [44] and STO [19]). Plor borrows these ideas to optimize throughput, but we consider tail latency as well. In timestamp ordering, Cicada [25] decides a transaction to commit or abort by comparing its timestamp with the read and write timestamps of the records it accessed. TicToc [55] does not grant timestamps for the running transaction in the read phase, instead, a valid timestamp is lazily computed in the commit phase. Plor uses timestamp as well, but with the main goal of ensuring that old transactions can always commit first. Neither Silo nor TicToc has such a guarantee – once a transaction aborts, it must use a newer timestamp for committing; differently, aborted transactions in Plor use old timestamps to increase their priority in the next commit.

Hybrid Concurrency Control. The idea of hybridizing OCC and 2PL has been implemented many times in the past [10, 39, 49]. For example, MOCC [49] uses 2PL and OCC dynamically to access records based on their hotness. It locks hot records via a variant of NO_WAIT to prevent writers from clobbering readers, and reads cold records optimistically like traditional OCC to avoid the unnecessary locking overhead. However, such a combination does not reduce tail latencies since neither NO_WAIT nor OCC prevents the aborted transactions from aborting again. Hsync [39] achieves high performance graph analytics through synchronous parallel processing by co-using OCC and 2PL. it uses a locking-based scheduler for high degree vertices, and a non-locking-based scheduler for low degree vertices. Callas [51] proposes the modular concurrency control that partitions transactions into groups and allows different concurrency control protocols to be used in different groups. Spanner [10] is a distributed database based on 2PL and it delays write-lock acquisition, which is similar to that of Plor. Polyjuice [48] decouples the timeline of a transaction into multiple steps and uses learning-based techniques to dynamically select the most optimal protocol for executing each step. CormCC [42] provides a framework for mixing different CC protocols and changing them online with minimal overhead. Above approaches follow a methodology of selecting a suitable protocol when a certain scenario is met, while the internal logic of each protocol is unchanged. Differently, Plor integrates the principles of OCC and WOUND_WAIT into a new protocol, whose main goal is considering both throughput and tail latency, which are mostly ignored by past work.

8 CONCLUSION

In this paper, we present *pessimistic locking and optimistic reading* (Plor), a new concurrency control scheme that enables both high throughput and low tail latency, and evaluate it under various setups. While the techniques inside Plor may not suitable for all kinds of setups, we find that Plor achieves close or comparable throughput as that of OCC and reduces 99.9th percentile latency by an order of magnitude when running in stored-procedure mode. Delaying write lock acquisition further improves throughput by 2× for interactive mode under YCSB-A workloads.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous reviewers for their feedback. This work is supported by the National Natural Science Foundation of China (Grant No. 61832011). Youmin Chen is also supported by the National Postdoctoral Program for Innovative Talents (Grant No. BX2021154), and the China Postdoctoral Science Foundation (Grant No. 2021M701887).

REFERENCES

[1] 2010. TPC benchmark C. "http://www.tpc.org/tpcc/".

[2] 2010. TPC benchmark E. "http://www.tpc.org/tpce/".

[3] 2020. A fast multi-producer, multi-consumer lock-free concurrent queue for C++11. "https://github.com/cameron314/concurrentqueue".

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 49–65.

[6] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter, and Siddhartha Sen. 2018. RobinHood: Tail Latency-Aware Caching—Dynamically Reallocating from Cache-Rich to Cache-Poor. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 195–212.

[7] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec. 1983), 465–483. https://doi.org/10.1145/319996.319998

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 251–264.

[11] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[12] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[13] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 79–94. https://www.usenix.org/conference/nsdi19/presentation/didona

[14] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 169–182. https://doi.org/10.14778/3282495.3282502

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[16] John Giacomoni, Tipp Moseley, and Manish Vachharajani. 2008. FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA) *(PPoPP '08)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/1345206.1345215

[17] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 161–175. https://doi.org/10.1145/2786763.2694384

[18] J.R. Haritsa, M.J. Carey, and M. Livny. 1990. Dynamic real-time optimistic concurrency control. In *[1990] Proceedings 11th Real-Time Systems Symposium*. 94–103. https://doi.org/10.1109/REAL.1990.128734

[19] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 629–642. https://doi.org/10.14778/3377369.3377373

[20] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for usecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[21] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[22] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1675–1687. https://doi.org/10.1145/2882903.2882905

[23] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 691–706. https://doi.org/10.1145/2723372.2746480

[24] Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. 2016. Work Stealing for Interactive Services to Meet Target Latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 13 pages. https://doi.org/10.1145/2851141.2851151

[25] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 21–35. https://doi.org/10.1145/3035918.3064015

[26] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[27] Shuai Mu, Sebastian Angel, and Dennis Shasha. 2019. Deferred Runtime Pipelining for Contentious Multicore Software Transactions. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 40, 16 pages. https://doi.org/10.1145/3302424.3303966

[28] Intel Newsroom. 2019. INTEL@ OPTANE^{TM} DC Persistent Memory. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[29] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[30] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 184–200. https://doi.org/10.1145/3132747.3132766

[31] Heidi Pan, Benjamin Hindman, and Krste Asanović. 2010. Composing Parallel Software Efficiently with Lithe. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 376–387. https://doi.org/10.1145/1806596.1806639

[32] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. https://www.cs.cmu.edu/~pavlo/slides/pavlo-keynote-sigmod2017.pdf.

[33] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. 2015. Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain) *(PODC '15)*. Association for Computing Machinery, New York, NY, USA, 217–226. https://doi.org/10.1145/2767386.2767438

[34] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX

Association, USA, 1–16.

[35] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[36] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[37] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostiundefined, and Sean Braithwaite. 2017. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 95–110. https://doi.org/10.1145/3064176.3064209

[38] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11 (July 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[39] Zechao Shang, Feifei Li, Jeffrey Xu Yu, Zhiwei Zhang, and Hong Cheng. 2016. Graph Analytics Through Fine-Grained Parallelism. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 463–478. https://doi.org/10.1145/2882903.2915238

[40] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.* 20, 3 (Sept. 1995), 325–363. https://doi.org/10.1145/211414.211427

[41] Mukesh Singhal. 1988. Issues and Approaches to Design of Real-Time Database Systems. *SIGMOD Rec.* 17, 1 (March 1988), 19–33. https://doi.org/10.1145/44203.44205

[42] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-Free and Reconfigurable Mixed Concurrency Control. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 809–822.

[43] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[44] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[45] Özgür Ulusoy and Geneva G. Belford. 1992. Concurrency Control in Real-Time Database Systems. In *Proceedings of the 1992 ACM Annual Conference on Communications* (Kansas City, Missouri, USA) *(CSC '92)*. Association for Computing Machinery, New York, NY, USA, 181–188. https://doi.org/10.1145/131214.131237

[46] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, USA, 363–378.

[47] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. 2003. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 268–281. https://doi.org/10.1145/945445.945471

[48] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 198–216. https://www.usenix.org/conference/osdi21/presentation/wang-jiachen

[49] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (Oct. 2016), 49–60. https://doi.org/10.14778/3015274.3015276

[50] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[51] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 279–294. https://doi.org/10.1145/2815400.2815430

[52] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[53] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[54] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[55] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935

[56] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for in-Memory Databases. *Proc. VLDB Endow.* 9, 6 (Jan. 2016), 504–515. https://doi.org/10.14778/2904121.2904126

[57] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. 2013. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/2517349.2522729